From logic programs updates to action description updates*

J. J. Alferes[1], F. Banti[1], and A. Brogi[2]

[1] CENTRIA, Universidade Nova de Lisboa, Portugal,
jja|banti@di.fct.unl.pt
[2] Dipartimento di Informatica, Università di Pisa, Italy,
brogi@di.unipi.it

Abstract. An important branch of investigation in the field of agents has been the definition of high level languages for representing effects of actions, the programs written in such languages being usually called action programs. Logic programming is an important area in the field of knowledge representation and some languages for specifying updates of Logic Programs had been defined. Starting from the update language Evolp, in this work we propose a new paradigm for reasoning about actions called Evolp action programs.
We provide translations of some of the most known action description languages into Evolp action programs, and underline some peculiar features of this newly defined paradigm. One such feature is that Evolp action programs can easily express changes in the rules of the domains, including rules describing changes.

1 Introduction

In the last years the concept of agent has become central in the field of Artificial Intelligence. "*An agent is just something that acts*" [26]. Given the importance of the concept, ways of representing actions and their effects on the environment have been studied. A branch of investigation in this topic has been the definition of high level languages for representing effects of actions [7, 12, 14, 15], the programs written in such languages being usually called *action programs*. Action programs specify which facts (or fluents) change in the environment after the execution of a set of actions. Several works exist on the relation between these action languages and Logic Programming (LP) (e.g. [5, 12, 21]). However, despite the fact that LP has been successfully used as a language for declaratively representing knowledge, the mentioned works basically use LP for providing an operational semantics, and implementation, for action programs. This is so because normal logic programs, and most of their extensions, have no in-built means for dealing with changes, something that is quite fundamental for action languages.

* This work was partially supported by project FLUX (POSI/40958/SRI/2001), and by the European Commission within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net).

In recent years some effort was devoted to explore and study the problem of how to update logic programs with new rules [3, 8, 10, 19, 20, 17]. Here, knowledge is conveyed by sequences of programs, where each program in a sequence is an update to the previous ones. For determining the meaning of sequences of logic programs, rules from previous programs are assumed to hold by inertia after the updates (given by subsequent programs) unless rejected by some later rule. LP update languages [2, 4, 9, 19], besides giving meaning to sequences of logic programs, also provide in-built mechanisms for constructing such sequences. In other words, LP update languages extend LP by providing means to specify and reason about rule updates.

In [5] the authors show, by examples, a possible use the LP update language LUPS [4] for representing effects of actions providing a hint for the possibility of using LP updates languages as an action description paradigm. However, the work done does not provide a clear view on how to use LP updates for representing actions, nor does it establishes an exact relationship between this new possibility and existing action languages. Thus, the eventual advantages of the LP update languages approach to actions are still not clear.

The present work tries to clarify these points. This is done by establishing a formal relationship between one LP update language, namely the Evolp language [2], and existing action languages, and by clarifying how to use this language for representing actions in general.

Our investigation starts by, on top of Evolp, defining a new action description language, called Evolp Action Programs (EAPs), as a macro language for Evolp. Before developing a complete framework for action description based on LP updates, in this work we focus on the basic problem in the field, i.e. the prediction of the possible future states of the world given a complete knowledge of the current state and the action performed. Our purpose is to check, already at this stage, the potentiality of an action description language based on the Evolp paradigm.

We then illustrate the usage of EAPs by an example involving a variant of the classical Yale Shooting Problem. An important point to clarify is the comparison of the expressive capabilities of the newly defined language with that of the existing paradigms. We consider the action languages $\mathcal{A}$ [12], $\mathcal{B}$ [13] (which is a subset of the language proposed in [14]), and (the definite fragment of) $\mathcal{C}$ [15]. We provides simple translations of such languages into EAPs, hence proving that EAPs are *at least as expressive* as the cited action languages.

Coming to this point, the next natural question is what are the possible advantages of EAPs. The underlying idea of action frameworks is to describe dynamic environments. This is usually done by describing rules that specify, given a set of external actions, how the environment evolves. In a dynamic environment, however, not only the facts but also the "rules of the game" can change, in particular *the rules describing the changes*. The capability of describing such kind of *meta level changes* is, in our opinion, an important feature of an action description language. This capability can be seen as an instance of *elaboration tolerance* i.e. *"the ability to accept changes to a person's or a computer's*

representation of facts about a subject without having to start all over" [25]. In [15] this capability is seen as a central point in the action descriptions field and the problem is addressed in the context of the $\mathcal{C}$ language. The final words of [15] are *"Finding ways to further increase the degree of elaboration tolerance of languages for describing actions is a topic of future work"*. We address this topic in the context of EAPs and show EAPs seem, in this sense, more flexible than other paradigms. Evolp provides specific commands that allow for the specification of updates to the initial program, but also provides the possibility to specify updates of these updates commands. We show, by successive elaborations of the Yale shooting problem, how to use this feature to describe updates of the problem that come along with the evolution of the environment.

The rest of the paper is structured as follows. In section 2 we review some background and notation. In section 3 we define the syntax and semantics of Evolp action programs, and we illustrate the usage of EAPs by an example involving a variant of the classical Yale Shooting Problem. In section 4 we establish the relationship between EAPs and the languages $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. In section 5 we discuss the possibility of updating the EAPs, and provide an example of such feature. Finally, in section 6, we conclude and trace a route for future developments. To facilitate the reading, and given that some of the results have proofs of some length, instead of presenting proofs along with the text, we expose them all in appendix A.

2 Background and notation

In this section we briefly recall syntax and semantics of *Dynamic Logic Programs* [1], and the syntax and semantics for Evolp [2]. We also recall some basic notions and notation for action description languages. For a more detailed background on action languages see e.g. [12].

2.1 Dynamic logic programs and Evolp

The main idea of logic programs updates is to update a logic program by another logic program or by a *sequence* of logic programs, also called *Dynamic Logic Programs* (DLPs). The initial program of a DLP corresponds to the initial knowledge of a given (dynamic) domain, and the subsequent ones to successive updates of the domain. To represent negative information in logic programs and their updates, following [3] we allow for default negation *not* A not only in the premises of rules but also in their heads i.e., we use generalized logic programs (GLPs) [22].

A language $\mathcal{L}$ is any set of propositional atoms. A literal in $\mathcal{L}$ is either an atom of $\mathcal{L}$ or the negation of an atom. In general, given any set of atoms $\mathcal{F}$, *we denote by* $\mathcal{F}_L$ *the set of literals over* $\mathcal{F}$. Given a literal F, if $F = Q$, where Q is an atom, by *not* F we denote the negative literal *not* Q. Viceversa, if $F =$ *not* Q, by *not* F we denote the atom Q. A GLP defined over a propositional language $\mathcal{L}$ is a set of rules of the form $F \leftarrow Body$, where F is a literal in $\mathcal{L}$, and $Body$

is a *set* of literals in $\mathcal{L}$.[1] An *interpretation* I over a language $\mathcal{L}$ is any set of literals in $\mathcal{L}$ such that, for each atom A, either $A \in I$ or *not* $A \in I$. We say a set of literals $Body$ is true in an interpretation I (or that I satisfies $Body$) iff $Body \subseteq I$. In this paper we will use programs containing variables. As usual when programming within the stable models semantics, a program with variables stands for the propositional program obtained as the set of all possible ground instantiations of the rules.

Two rules τ and η are *conflicting* (denoted by $\tau \bowtie \eta$) iff the head of τ is the atom A and the head of η is *not* A, or viceversa. A Dynamic Logic Program over a language $\mathcal{L}$ is a sequence $P_1 \oplus \ldots \oplus P_m$ (also denoted $\oplus P_i^m$) where the P_is are GLPs defined over $\mathcal{L}$. The *refined stable model semantics* of such a DLP, defined in [1], assigns to each sequence $P_1 \oplus \ldots \oplus P_n$ a set of stable models (that is proven there to coincide with the stable models semantics when the sequence is formed by a single normal [11] or generalized program [22]). The rationale for the definition of a stable model M of a DLP is made in accordance with the *causal rejection principle* [10, 19]: If the body of a rule in a given update is true in M, then that rule rejects all rules in previous updates that are conflicting with it. Such rejected rules are ignored in the computation of the stable model. In the refined semantics for DLPs a rule may also reject conflicting rules that belong to the same update. Formally, the set of rejected rules of a DLP $\oplus P_i^m$ given an interpretation M is:

$$Rej^S(\oplus P_i^m, M) = \{\tau \mid \tau \in P_i : \ \exists\, \eta \in P_j \ i \leq j, \ \tau \bowtie \eta \ \wedge \ Body(\eta) \subseteq M\}$$

Moreover, an atom A is assumed false by default if there is no rule, in none of the programs in the DLP, with head A and a true body in the interpretation M. Formally:

$$Default(\oplus P_i^m, M) = \{not\ A \mid \ \nexists\ A \leftarrow Body \in \bigcup P_i \quad \wedge Body \subseteq M\}$$

If $\oplus P_i^m$ is clear from the context, we omit it as first argument of the above functions.

Definition 1. *Let $\oplus P_i^m$ be a DLP over language $\mathcal{L}$ and M an interpretation. M is a refined stable model of $\oplus P_i^m$ iff*

$$M = least\left(\left(\bigcup_i P_i \setminus Rej^S(M)\right) \cup Default(M)\right)$$

where $least(P)$ denotes the least Herbrand model of the definite program [23] obtained by considering each negative literal not A in P as a new atom.

Having defined the meaning of sequences of programs, we are left with the problem of how to come up with those sequences. This is the subject of LP update

[1] Note that, by defining rule bodies as sets, the order and number of occurrences of literals do not matter.

languages [2, 4, 9, 19]. Among the existing languages, Evolp [2] uses a particulary simple syntax, which extends the usual syntax of GLPs by introducing the special predicate $assert/1$. Given any language $\mathcal{L}$, the language $\mathcal{L}_{assert}$ is recursively defined as follows: every atom in $\mathcal{L}$ is also in $\mathcal{L}_{assert}$; for any rule τ over $\mathcal{L}_{assert}$, the atom $assert(\tau)$ is in $\mathcal{L}_{assert}$; nothing else is in $\mathcal{L}_{assert}$. An *Evolp program* over $\mathcal{L}$ is any GLP over $\mathcal{L}_{assert}$. An *Evolp sequence* is a sequence (or DLP) of Evolp programs. The rules of an Evolp program are called *Evolp rules*.

Intuitively an expression $assert(\tau)$ stands for "update the program with the rule τ". Notice the possibility in the language to nest an assert expression in another. The intuition behind the Evolp semantics is quite simple. Starting from the initial Evolp sequence $\oplus P_i^m$ we compute the set, $\mathcal{SM}(\oplus P_i^m)$, of the stable models of $\oplus P_i^m$. Then, for any element M in $\mathcal{SM}(\oplus P_i^m)$, we update the initial sequence with the program P_{m+1} consisting of the set of rules τ such that the atom $assert(\tau)$ belongs to M. In this way we obtain the sequence $\oplus P_i^m \oplus P_{m+1}$. Since $\mathcal{SM}(\oplus P_i^m)$ contains, in general, several models we may have different lines of evolution. The process continues by obtaining the various $\mathcal{SM}(\oplus P_i^{m+1})$ and, with them, various $\oplus P_i^{m+2}$. Intuitively, the program starts at step 1 already containing the sequence $\oplus P_i^m$. Then it updates itself with the rules asserted at step 1, thus obtaining step 2. Then, again, it updates itself with the rules asserted at this step, and so on. The evolution of any Evolp sequence can also be influenced by external events. An external event is itself an Evolp program. If, at a given step n, the programs receives the external update E_n, the rules in E_n are added to the last self update for the purpose of computing the stable models determining the next evolution but, in the successive step $n+1$ they are no longer considered (that's why they are called *events*). Formally:

Definition 2. *Let n and m be natural numbers. An* evolution interpretation *of length n, of an evolving logic program $\oplus P_i^m$ is any finite sequence $\mathcal{M} = M_1, \ldots, M_n$ of interpretations over $\mathcal{L}_{assert}$. The* evolution trace *associated with $\mathcal{M}$ and $\oplus P_i^m$ is the sequence $P_1 \oplus \ldots P_m \oplus P_{m+1} \ldots \oplus P_{m+n-1}$, where, for $1 \leq i < n$*

$$P_{m+i} = \{\tau \mid assert(\tau) \in M_{m+i-1}\}$$

Definition 3 (Evolving stable models). *Let $\oplus P_i^m$ and $\oplus E_i^n$ be any Evolp sequences, and $\mathcal{M} = M_1, \ldots, M_n$ be an evolving interpretation of length n. Let $P_1 \oplus \ldots \oplus P_{m+n-1}$ be the evolution trace associated with $\mathcal{M}$ and $\oplus P_i^m$. We say that $\mathcal{M}$ is an evolving stable model of $\oplus P_i^m$ with event sequence $\oplus E_i^n$ at step n iff M_k is a refined stable model of the program $P_1 \oplus \ldots \oplus (P_k \cup E_k)$ for any k, with $m \leq k \leq m+n-1$.*

2.2 Action languages

The purpose of an action language is to provide ways of describing how an environment evolves given a set of external actions. A specific environment that can be modified through external actions is called an *action domain*. To any action domain we associate a pair of sets of atoms $\mathcal{F}$ and $\mathcal{A}$. We call the elements

of $\mathcal{F}$ *fluent atoms* or simply *fluents*, and the elements of $\mathcal{A}$ *action atoms* or simply *actions*. Basically, the fluents are the observables in the environment and the actions are, clearly, the external actions. A *fluent literal* (resp. *action literal*) is an element of $\mathcal{F}_L$ (resp. an element of $\mathcal{A}_L$). In the following, we will use the letter Q to denote a fluent atom, the letter F to denote a fluent literal, and the letter A to denote an action atom. A *state of the world* (or simply a *state*) is any interpretation over $\mathcal{F}$. We say a fluent literal F is true at a given state s iff F belongs to s. Given a set (or, by abuse of notation, a conjunction) of fluent literals $Cond$ we say s *satisfies* $Cond$, and write $s \models Cond$, iff $Cond \subseteq s$.

Each action language provides ways to describe action domains through sets of expression called *action programs*. Usually, the semantics of an action program is defined in terms of a *transition system*, i.e. a function whose argument is any pair (s, K), where s is a state of the world and K is a subset of $\mathcal{A}$, and whose value is any set of states of the world. Intuitively, given the current state of the world, a transition system specifies which are the possible resulting states after simultaneously performing all actions in K.

Two kinds of expressions that are common within action description languages are *static* and *dynamic rules*. The *static rules* basically describe the rules of the domain, while *dynamic rules* describe effects of actions. A dynamic rule has a set of *preconditions*, namely conditions that have to be satisfied in the present state in order to have a particular effect in the future state, and *post-conditions* describing such an effect.

In the following we will consider three existing action languages, namely: $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. The language $\mathcal{A}$ [13] is very simple. It only allows dynamic rules of the form

$$A \textbf{ causes } F \textbf{ if } Cond$$

where $Cond$ is a conjunction of fluent literals. Such a rule intuitively means: performing the action A causes F to be true in the next state if $Cond$ is true in the current state. The language $\mathcal{B}$ [13] is an extension of $\mathcal{A}$ which also considers static rules. In $\mathcal{B}$, static rules are expressions of the form

$$F \textbf{ if } Body$$

where $Body$ is a conjunction of fluent literals. Intuitively, such a rule means: if $Body$ is true in the current state, then F is also true in the current state. A fundamental notion, in both $\mathcal{A}$ and $\mathcal{B}$, is *fluent inertia* [13]. A fluent F is inertial if its truth value is preserved from a state to another, unless it is changed by the (direct or indirect) effect of an action. Hereafter a program written in the language $\mathcal{B}$ will be called a $\mathcal{B}$ program.

The semantics of $\mathcal{B}$ is defined in terms of a transition system, as sketched above. For introducing the particular transition function that, given a state s and a set of actions K, determines the possible resulting states according to $\mathcal{B}$, we first consider the set $D(s, K)$ of fluents literals that are true as a (direct) consequence of actions. Any literal F is a direct consequence of state s and actions K if it is in the head of a dynamic rule A **causes** F **if** $Cond$ such that $A \in K$ and $Cond$ is true in s. Then a state s' is a possible resulting states from

s iff any fluent literal in s is an element of $D(s, K)$ or is a true literal in s (that followed by inertia) or is a consequence of a static rule:

Definition 4. *Let P be any $\mathcal{B}$ program with set of fluents $\mathcal{F}$, let $\mathcal{R}$ be the set of all static rules in P, and let s be a state and K any set of actions. Moreover, let $D(s, K)$ be the following set of literals*

$$D(s, K) = \{F : \exists\ A\ \textbf{\textit{causes}}\ F\ \textbf{\textit{if}}\ Cond\ \in\ P\ s.t.\ A \in K\ \wedge\ s \models Cond\}$$

and let $\mathcal{R}^{LP}$ be the logic program:

$$\mathcal{R}^{LP} = \{F \leftarrow Body : F\ \textbf{\textit{if}}\ Body \in \mathcal{R}\}$$

A state s' is a resulting state from s given P and the set of actions K iff

$$s' = least(s \cap s' \cup D(s, K) \cup \mathcal{R}^{LP})$$

where $least(P)$ is as in Definition 1

For a detailed explanation of $\mathcal{A}$ and $\mathcal{B}$ see e.g. [13].

Static and dynamic rules are also the ingredients of the action language $\mathcal{C}$ [15, 16]. Static rules in $\mathcal{C}$ are of the form

caused J **if** H

while dynamic rules are of the form

caused J **if** H **after** O

where J and H are formulae such that any literal in them is a fluent literal, and O is any formula such that any literal in it is a fluent or an action literal. The formula O is the precondition of the dynamic rule and the static rule **caused** J **if** H is its postcondition. The semantic of $\mathcal{C}$ is based on *causal theories*[15]. Causal theories are sets of rules of the form **caused** J **if** H, each such rule meaning: If H is true this is an explanation for J. A basic principle of causal theories is that something is true iff it is caused by something else. Given any action program P, a state s and a set of actions K, we consider the causal theory T given by the static rules of P and the postconditions of the dynamic rules whose preconditions are true in $s \cup K$. Then s' is a possible resulting state iff it is a causal model of T.

3 Evolp action programs

As we have seen, Evolp and action description languages share the idea of a system that evolves. In both, the evolution is influenced by external events (respectively, updates and actions). Evolp is actually a programming language devised for representing any kind of computational problem, while action description languages are devised for the specific purpose of describing actions. A natural

idea is then to develop special kind of Evolp sequences for representing actions, and then compare such kind of programs with existing action description languages. We will develop one such kind of programs, and call them *Evolp Action Programs* (EAPs).

Following the underlying notions of Evolp, we use the basic construct *assert* for defining special-purpose macros. As it happens with other action description languages, EAPs are defined over a set of fluents $\mathcal{F}$ and a set of actions $\mathcal{A}$. In EAPs, a state of the world is any interpretation over $\mathcal{F}$. To describe action domains we use an initial Evolp sequence, $I \oplus D$. The Evolp program D contains the description of the environment, while I contains some initial declarations, as it will be clarified later. As in $\mathcal{B}$ and $\mathcal{C}$, EAPs contain static and dynamic rules.

A *static rule* over $(\mathcal{F}, \mathcal{A})$ is simply an Evolp rule of the form

$$F \leftarrow Body.$$

where F is a fluent literal and $Body$ is a set of fluent literals.

A *dynamic rule* over $(\mathcal{F}, \mathcal{A})$ is a (macro) expression

$$\mathbf{effect}(\tau) \leftarrow Cond.$$

where τ is any static rule $F \leftarrow Body$ and $Cond$ is any set of fluent or action literals. The intuitive meaning of such a rule is that the static rule τ has to be considered *only* in those states whose predecessor satisfies condition $Cond$. Since some of the conditions literals in $Cond$ may be action atoms, such a rule may describe the effect of a given set of actions under some conditions. Such an expression stands for the following set of Evolp rules:

$$F \leftarrow Body,\ event(F \leftarrow Body). \tag{1}$$

$$assert(event(F \leftarrow Body)) \leftarrow Cond. \tag{2}$$

$$assert(not\ event(F \leftarrow Body)) \leftarrow event(\tau), not\ assert(event(F \leftarrow Body)) \tag{3}$$

where $event(F \leftarrow Body)$ is a new literal. Let us see how the above set of rules fits with its intended intuitive meaning. Rule (1) is not applicable whenever $event(F \leftarrow Body)$ is false. If at some step n, the conditions $Cond$ are satisfied, then, by rule (2), $event(F \leftarrow Body)$ becomes true at step $n+1$. Hence, at step $n+1$, rule (1) will play the same role as static rule $F \leftarrow Body$. If at step $n+1$ $Cond$ is no longer satisfied, then, by rule (3) the literal $event(F \leftarrow Body)$ will become false again, and then rule (1) will be again not effective.

Besides static and dynamic rules, we still need another ingredient to complete our construction. As we have seen in the description of the $\mathcal{B}$ language, a notable concept is fluent inertia. This idea is not explicit in Evolp where *the rules* (and not the fluents) are preserved by inertia. Nevertheless, we can show how to obtain fluent inertia by using macro programming in Evolp. An *inertial declaration* over $(\mathcal{F}, \mathcal{A})$ is a (macro) expression $\mathbf{inertial}(\mathcal{K})$, where $\mathcal{K} \subseteq \mathcal{F}$. The intended intuitive meaning of such an expression is that the fluents in $\mathcal{K}$ are inertial. Before defining what this expression stands for, we state that the above mentioned program I is always of the form $\mathbf{initialize}(\mathcal{F})$, where $\mathbf{initialize}(\mathcal{F})$ stands for the set of

rules $Q \leftarrow prev(Q)$, where Q is any fluent in $\mathcal{F}$, and $prev(Q)$ are new atoms not in $\mathcal{F} \cup \mathcal{A}$. The *inertial declaration* **inertial($\mathcal{K}$)** stands for the set (where Q ranges over $\mathcal{K}$):

$$assert(prev(Q)) \leftarrow Q. \qquad assert(not\ prev(Q)) \leftarrow not\ Q.$$

Let us consider the behaviour of this macro. If we do not declare Q as an inertial fluent, the rule $Q \leftarrow prev(Q)$ has no effect. If we declare Q as an inertial literal, $prev(Q)$ is true in the current state iff in the previous state Q was true. Hence, in this case, Q is true in the current state *unless* there is a static or dynamic rule that rejects such assumption. Viceversa, if Q was false in the previous state, then Q is true in the current one iff it is derived by a static or dynamic rule. We are now ready to formalize the syntax of Evolp action programs.

Definition 5. *Let $\mathcal{F}$ and $\mathcal{A}$ be two disjoint sets of propositional atoms. An Evolp action program (EAP) over $(\mathcal{F},\ \mathcal{A})$ is any Evolp sequence $I \oplus D$, where $I = Initialize(\mathcal{F})$, and D is any set with static and dynamic rules, and inertial declarations over $(\mathcal{F}, \mathcal{A})$*

Given an Evolp action program $I \oplus D$, the initial state of the world s (which, as stated above, is an interpretation over $\mathcal{F}$) is passed to the program together with the set K of the actions performed at s, as part of an external event. A resulting state is the last element of any evolving stable model of $I \oplus D$ given the event $s \cup K$ restricted to the set of fluent literals. I.e:

Definition 6. *Let $I \oplus D$ be any EAP over $(\mathcal{F}, \mathcal{A})$, and s a state of the world. Then s' is a resulting state from s given $I \oplus D$ and the set of actions K iff there exists an evolving stable model M_1, M_2 of $I \oplus D$ given the external events $s \cup K, \emptyset$ such that $s' \equiv_{\mathcal{F}} M_2$ (where by $s' \equiv_{\mathcal{F}} \mathcal{M}_2$ we simply mean $s' \cap \mathcal{F}_{Lit} = M_2 \cap \mathcal{F}_{Lit}$).*

This definition can be easily generalized to sequences of set of actions.

Definition 7. *Let $I \oplus D$ be any EAP and s a state of the world. Then s' is a resulting state from s given $I \oplus D$ and the sequence of sets of actions $K_1 \ldots,\ K_n$ iff there exists an evolving stable model $M_1, \ldots, M_{n+1}$ of $I \oplus D$ given the external events $(s \cup K_1), \ldots,\ K_n, \emptyset$ such that $s' \equiv_{\mathcal{F}} M_{n+1}$.*

Since EAPs are based on the Evolp semantics, which in turn is an extension of the stable model semantics for normal logic programs, we can easily prove that the complexity of the computation of the two semantics is the same.

Theorem 1. *Let $I \oplus D$ be any EAP over $(\mathcal{F}, \mathcal{A})$, s a state of the world and $K \subseteq \mathcal{A}$. To find a resulting state s' from s given $I \oplus D$ and the set of actions K is an NP-complete problem.*

It is important to notice that, if the initial state s does not satisfies the static rules of the EAP, the correspondent Evolp sequence has no stable model, and hence there will be no successor state. This is, in our opinion, a good result: The

initial state is just a state as any other. It would be strange if such state would not satisfy the rules of the domain. If this situation occurs, most likely either the translation of the rules, or the one of the state, presents some errors. From now onwards we will assume that the initial state satisfies the static rules of the domain.

To illustrate EAPs, we now show an example of their usage by elaborating on probably the most famous example of reasoning about actions. The presented elaboration highlights some important features of EAPs, viz. the possibility of handling non-deterministic effects of actions, non-inertial fluents, non-executable actions, and effects of actions lasting for just one state.

Example 1 (An elaboration of the Yale shooting problem). In the original Yale shooting problem [27], there is a single-shot gun which is initially unloaded, and a turkey which is initially alive. One can load the gun and shoot the turkey. If one shoots, the gun becomes unloaded and the turkey dies. We consider a slightly more complex scenario where there are several turkeys, and where the shooting action refers to a specific turkey. Each time one shoots as specific turkey, one either hits and kills the bird, or misses it. Moreover, the gun becomes unloaded and there is a bang. It is not possible to shoot with an unloaded gun. We also add the property that any turkey moves iff it is not dead.

For expressing that an action is not executable under some conditions, we make use of a well known behaviour of the stable model semantics. Suppose a given EAP contains a dynamic rules of the form $\mathbf{effect}(u \leftarrow not\ u) \leftarrow Cond$, where u is a literal which does not appear elsewhere (in the following, for representing such rules, we use the notation $\mathbf{effect}(\bot) \leftarrow Cond$). With such a rule, if $Cond$ is true in the current state, then there is no resulting state. This happens because, as it is well known, programs containing $u \leftarrow not\ u$ and no other rules for u, have no stable models.

To represent the problem, we consider the fluents $dead(X)$, $moving(X)$, $hit(X)$, $missed(X)$, $loaded$, $bang$, plus the auxiliary fluent u, and the actions $shoot(X)$ and $load$ (where the Xs range over the various turkeys). The fluents $dead(X)$ and $loaded$ are inertial fluents, since their truth value should remain unchanged until modified by some action effect. The fluents $missed(X)$, $hit(X)$ and $bang$ are not inertial. The problem is encoded by the EAP $I \oplus D$, where

$$I = \mathbf{initialize}(dead(X), moving(X), missed(X), hit(X),\ loaded,\ bang,\ u)$$

and D is the following set of expressions

$\mathbf{effect}(\bot) \leftarrow shoot(X),\ not\ loaded$	$\mathbf{inertial}(loaded)$
$moving(X) \leftarrow not\ dead(X)$	$\mathbf{inertial}(dead(X))$
$\mathbf{effect}(dead(X) \leftarrow hit(X)) \leftarrow\ shoot(X)$	$\mathbf{effect}(loaded) \leftarrow load$
$\mathbf{effect}(hit(X) \leftarrow not\ missed(X)) \leftarrow shoot(X)$	$\mathbf{effect}(bang) \leftarrow shoot(X)$
$\mathbf{effect}(missed(X) \leftarrow not\ hit(X)) \leftarrow shoot(X)$	$\mathbf{effect}(not\ loaded) \leftarrow shoot(X)$

Let us analyze this EAP. The first rule encodes the impossibility to execute the action $shoot(X)$ when the gun is unloaded. The static rule $moving(X) \leftarrow$

not $dead(X)$ implies that, for any turkey X, $moving(X)$ is true if $dead(X)$ is false. Since this is the only rule for $moving(X)$, it further holds that $moving(X)$ is true iff $dead(X)$ is false. Notice that declaring $moving(tk)$ as inertial, would result, in our description, in the possibility of having a moving dead turkey! This is why fluents $moving(X)$ have not been declared as inertial. In fact, suppose we insert **inertial(**$moving(X)$**)** in the EAP above. Suppose further that $moving(tk)$ is true at state s, that one shoots at tk and kills it. Since $moving(tk)$ is an inertial fluent, in the resulting state $dead(tk)$ is true, but $moving(tk)$ remains true by inertia. Also notable is how effects that last only for one state, like the noise provoked by the shoot, are easily encoded. The last three dynamic rules on the left encode a non deterministic behaviour: each shoot action can either hit and kill a turkey, or miss it.

To see how this EAP encodes the desired behaviour of this domain, consider the following example of evolution. In this example, to lightening the notation, we omit the negative literals belonging to interpretations. Let us consider the initial state {} (which means that all fluents are false). The state will remain unchanged until some action is performed. If one load the gun, the program is updated with the external event $\{load\}$. In the unique successor state, the fluent *loaded* is true and nothing else changes. The truth value of *loaded* remains then unchanged (by inertia) until some other action is performed. The same applies to fluents $dead(X)$. The fluents *bang*, $missed(X)$, and $hit(X)$ remain false by default. If one shoots at a specific turkey, say Smith, and the program is updated with the event $shoot(smith)$, several things happen. First, *loaded* becomes false, and *bang* becomes true, as an effect of the action. Moreover, the rules:

$$hit(smith) \leftarrow not\ missed(smith).$$
$$missed(smith) \leftarrow not\ hit(smith).$$
$$dead(smith) \leftarrow hit(smith).$$

are considered as rules of the domain for one state. As a consequence, there are two possible resulting states. In the first one, $missed(smith)$ is true, and all the others fluents are false. In the second one $hit(smith)$ is true, $missed(smith)$ is false and, by the rule $dead(smith) \leftarrow hit(smith)$, the fluent $dead(smith)$ becomes true. In both the resulting states, nothing happens to the truth value of the fluents $dead(X)$, $hit(X)$, and $dead(X)$ for $X \neq smith$.

4 Relationship to existing action languages

In this section we show embeddings into EAPs of the action languages $\mathcal{B}$ and (the definite fragment of) $\mathcal{C}^2$. We will assume that the considered initial states are consistent wrt. the static rules of the program, i.e. if the body of a static rule is true in the considered state, the head is true as well.

[2] The embedding of language $\mathcal{A}$ is not explicitly exposed here since $\mathcal{A}$ is a (proper) subset of the $\mathcal{B}$ language.

Let us consider first the $\mathcal{B}$ language. The basic ideas of static and dynamic rules are very similar in $\mathcal{B}$ and in EAPs. The main difference between the two is that in $\mathcal{B}$ *all* the fluents are inertial, whilst in EAPs only those that are declared as such are inertial. The translation of $\mathcal{B}$ into EAPs is then straightforward: All fluents are declared as inertial and then the syntax of static and dynamic rules is adapted. In the following we use, with abuse of notation, *Body* and *Cond* both for conjunctions of literals and for sets of literals.

Definition 8. *Let P be any action program in $\mathcal{B}$ with set of fluents $\mathcal{F}$. The translation $B(P, \mathcal{F})$ is the pair $(I^B \oplus D^{BP}, \mathcal{F}^B)$ where: $\mathcal{F}^B \equiv \mathcal{F}$, $I^B =$ **initialize($\mathcal{F}$)** and D^{BP} contains exactly the following rules:*

- ***inertial(Q)** for each fluent $Q \in \mathcal{F}$*
- *a rule $F \leftarrow Body$ for any static rule F **if** $Body$ in P.*
- *a rule **effect(F)** $\leftarrow A, Cond.$ for any dynamic rule A **causes** F **if** $Cond$ in P.*

Theorem 2. *Let P be any $\mathcal{B}$ program with set of fluents $\mathcal{F}$, $(I^B \oplus D^{BP}, \mathcal{F})$ its translation, s a state and K any set of actions. Then s' is a resulting state from s given P and the set of actions K iff it is a resulting state from s given $I^B \oplus D^{BP}$ and the set of actions K.*

This theorem makes it clear that there is a close relationship between *EAPs* and the $\mathcal{B}$ language. In practice, *EAPs* generalize $\mathcal{B}$ by allowing both inertial and non inertial fluents and by admitting rules, rather then simply facts, as effects of actions.

Let us consider now the action language $\mathcal{C}$. Given a complete description of the current state of the world and performed actions, the problem of finding a resulting state is a problem of the satisfiability of a causal theory, which is known to be $\sum_P^2$-hard (cf. [15]). So, this language belongs to a category with higher complexity than EAPs whose satisfiability is NP-complete. However, only a fragment of $\mathcal{C}$ is implemented and the complexity of such fragment is NP. This fragment is known as the *definite fragment* of $\mathcal{C}$ [15]. In this fragment, static rules are expressions of the form **caused** F **if** *Body* where F is a fluent literal and *Body* is a conjunction of fluent literals, while dynamic rules are expressions of the form **caused** *not* F **if** *Body* **after** *Cond* where *Cond* is a conjunction of fluent or action literals[3]. For this fragment it is possible to provide a translation into EAPs.

The main problem of the translation of $\mathcal{C}$ into EAPs lies in the simulation of causal reasoning with stable model semantics. The approach followed here to encode causal reasoning with stable models is in line with the one proposed in [21]. We need to introduce some auxiliary predicates and define a syntactic

[3] The definite fragment defined in [15] is (apparently) more general, allowing *Cond* and *Body* to be arbitrary formulae. However, it is easy to prove that such kind of expressions are equivalent to a set of expressions of the form described above

transformation of rules. Let $\mathcal{F}$ be a set of fluents, and let $\mathcal{F}^C$ denote the set of fluents $\mathcal{F} \cup \{Q_N \mid Q \in \mathcal{F}\}$. We add, for each $Q \in \mathcal{F}$, the constraints:

$$\leftarrow not\ Q, not\ Q_N. \tag{4}$$
$$\leftarrow Q,\ Q_N. \tag{5}$$

Let Q be a fluent and $Body = F_1, \ldots, F_n$ a conjunction of fluent literals. We will use the following notation: $\overline{Q} = not\ Q_N$, $\overline{not\ Q} = not\ Q$ and $\overline{Body} = \overline{F_1}, \ldots, \overline{F_n}$

Definition 9. *Let P be any action program in the definite fragment of $\mathcal{C}$ with set of fluents $\mathcal{F}$. The translation $C(P, \mathcal{F})$ is the pair $(I^C \oplus D^{CP}, \mathcal{F}^C)$ where: $\mathcal{F}^C$ is defined as above, $I^C \equiv$ **initialize**$(\mathcal{F}^C)$ and D^{CP} consists exactly of the following rules:*

- *a rule **effect**$(Q \leftarrow \overline{Body}) \leftarrow Cond$, for any dynamic rule in P of the form **caused** Q **if** $Body$ **after** $Cond$;*
- *a rule **effect**$(Q_N \leftarrow \overline{Body}) \leftarrow Cond$, for any dynamic rule in P of the form **caused** $not\ Q$ **if** $Body$ **after** $Cond$;*
- *a rule $Q \leftarrow \overline{Body}$, for any static rule in P of the form **caused** Q **if** $Body$;*
- *a rule $Q_N \leftarrow \overline{Body}$, for a static rule in P of the form **caused** $not\ Q$ **if** $Body$;*
- *The rules (4) and (5), for each fluent $Q \in \mathcal{F}$.*

For this translation we obtain a result similar to the one obtained for the translations of the $\mathcal{B}$ language:

Theorem 3. *Let P be any action program in the definite fragment of $\mathcal{C}$ with set of fluents $\mathcal{F}$, $(I^C \oplus D^{CP}, \mathcal{F}^C)$ its translation, s a state, s^C the interpretation over $\mathcal{F}^C$ defined as follows: $s^C = s \cup \{Q_N \mid Q \in s\} \cup \{not\ Q_N \mid not\ Q \in s\}$ and K any set of actions. Then s^* is a resulting state from s^C given $I^C \oplus D^{CP}$ and the set of actions K iff there exists s' such that s' is a resulting state from s, given P and the set K and $s^* \equiv_{\mathcal{F}_L} s'$.*

By showing translations of the action languages $\mathcal{B}$ and the definite fragment of $\mathcal{C}$ into EAPs, we proved that EAPs are *at least as expressive* as such languages. Moreover, the translations above are quite simple: basically one EAP static or dynamic rule for each static or dynamic rule in the other languages. The next natural question is: Are they *more expressive*?

5 Updates of action domains

Action description languages describe the rules governing a domain where actions are performed, and the environment changes. In practical situations, it may happen that the very rules of the domain change with time too. When this happens, it would be desirable to have ways of specifying the necessary updates to the considered action program, rather than to have to write a new one. EAPs are just a particular kind of Evolp sequences. So, as in general Evolp sequences, they can be updated by external events.

When one wants to update the existing rules with a rule τ, all that has to be done is to add the fact $assert(\tau)$ as an external event. This way, the rule τ is asserted and the existing Evolp sequence is updated. Following this line, we extend EAPs by allowing the external events to contain facts of the form $assert(\tau)$, where τ is an Evolp rule, and we show how they can be used to express updates to EAPs. For simplicity, below we use the notation $assert(R)$, where R is a set of rules, for the set of expressions $assert(\tau)$ where $\tau \in R$.

To illustrate how to update an EAP, we come back to Example 1. Let $I \oplus D$ be the EAP defined in there. Let us now consider that after some shots, and dead turkeys, rubber bullets are acquired. One can now either load the gun with normal bullets or with a rubber bullets, but not with both. If one shoots with a rubber loaded gun, the turkey is not killed.

To describe this change in the domain, we introduce a new inertial fluent representing the gun being loaded with rubber bullets. We have to express that, if the gun is rubber-loaded, one can not kill the turkey. For this purpose we introduce the new macro:

$$not\ \mathbf{effect}(F \leftarrow Body) \leftarrow Cond.$$

where F, is a fluent literal, $Body$ is a set of fluents literals and $Cond$ is a set of fluent or action literals. We refer to such expressions as *effects inhibitions*. This macro simply stands for the rule

$$assert(not\ event(F \leftarrow Body)) \leftarrow Cond.$$

where $event(F \leftarrow Body)$ is as before. The intuitive meaning is that, if the condition $Cond$ is true in the current state, any dynamic rule whose effect is the rule $F \leftarrow Body$ is ignored.

To encode the changes described above, we update the EAP with the external event E_1 consisting of the facts $assert(I_1)$ where

$$I_1 = (\mathbf{initialize}(rubber\_loaded))$$

Then, in the subsequent state, we update the program with the external update $E_2 = assert(D_1)$ where D_1 is the set of rules[4]

$$\begin{array}{l}
\mathbf{inertial}(rubber_loaded).\\
\mathbf{effect}(rubber_loaded) \leftarrow rubber_load.\\
\mathbf{effect}(not\ rubber_loaded) \leftarrow shoot(X).\\
\mathbf{effect}(\bot) \leftarrow rubber_loaded,\ load.\\
\mathbf{effect}(\bot) \leftarrow loaded,\ rubber_load.\\
not\ \mathbf{effect}(dead(X) \leftarrow hit(X)) \leftarrow rubber_loaded.
\end{array}$$

Let us analyze the proposed update. First, the fluent $rubber\_loaded$ is initialized. It is important to initialize any fluent before starting to use it. The

[4] In the remainder, we use $assert(U)$, where U is a set of macros (which are themselves sets of Evolp rules), to denote the set of all facts $assert(\tau)$ such that there exists a macro η in U with $\tau \in \eta$.

newly introduced fluent is declared as inertial, and two dynamic rules are added specifying that load actions are not executable when the gun is already loaded in a different way. Finally we use the new command to specify that the effect $dead(X) \leftarrow hit(X)$ does not occurs if, in the previous state, the gun was loaded with rubber bullets. Since this update is more recent than the original rule $\mathbf{effect}(dead(X) \leftarrow hit(X)) \leftarrow shoot(X)$, the dynamic rule is updated.

Basically updating the original EAP with the rule

$$not\ \mathbf{effect}(dead(X) \leftarrow hit(X)) \leftarrow rubber\_loaded.$$

has the effect of adding $not\ rubber\_loaded$ to the preconditions of the dynamic rule

$$\mathbf{effect}(dead(X) \leftarrow hit(X)) \leftarrow shoot(X).$$

So far we have shown how to update the preconditions of a dynamic rule. It is also possible to update static rules and the descriptions of effects of actions. Suppose the cylinder of the gun becomes dirty and, whenever one shoots, the gun may either work properly or fail. If the gun fails, the action *shoot* has no effect. We introduce two new fluents in the program with the event $assert(I_2)$ where $I_2 = \mathbf{initialize}(fails, work)$). Then, we assert the event $E_2 = assert(D_2)$ where D_2 is the following EAP

$$\begin{aligned}
\mathbf{effect}(fails \leftarrow not\ work) &\leftarrow shoot(X).\\
\mathbf{effect}(work \leftarrow not\ fails) &\leftarrow shoot(X).\\
not\ missed(X) &\leftarrow fails.\\
not\ hit(X) &\leftarrow fails.\\
not\ bang &\leftarrow fails.\\
\mathbf{effect}(loaded \leftarrow fails) &\leftarrow loaded.\\
\mathbf{effect}(rubber_loaded \leftarrow fails) &\leftarrow rubber_loaded.
\end{aligned}$$

The first two dynamic rules simply introduce the possibility that a failure may occur every time we shoot. The three static rules describe changes in the behaviour of the environment when the gun fails, and amount to negate what was entailed by static and dynamic rules in D. The last two dynamic rules update two of the dynamic rules in D and D_1, respectively. These rules specify that, when a failure occurs, the gun remain loaded with the same kind of bullet. Since the new rules of D_2 are more recent than the rules in D and D_1, they update these latter ones.

This last example shows how to update static and dynamic rules with new static and dynamic rules. To illustrate how this is indeed achieved in this example, we now show a possible evolution of the updated system. Suppose currently the gun is not loaded. One loads the gun with a rubber bullet, and then shoots at the turkey named Trevor. The initial state is $\{\}$. The first set of actions is $\{rubber\_load\}$ The resulting state after this action is $s' \equiv \{rubber\_loaded\}$. Suppose one performs the action *load*. Since the EAP is updated with the dynamic rule $\mathbf{effect}(\bot) \leftarrow rubber\_loaded,\ load.$ there is no resulting state. This happens because we have performed a non executable action. Suppose, instead, that the

second set of actions is $\{shoot(trevor)\}$. In this case there are three possible resulting states. In one the gun fails and, in it, the resulting state is again s'. In the second, the gun works but the bullet misses Trevor. In this case, the resulting state is $s''_1 \equiv \{missed(trevor)\}$. Finally, in the third, the gun works and the bullet hits Trevor. Since the bullet is a rubber bullet, Trevor is still alive. In this case the resulting state is $s''_2 \equiv \{hit(trevor)\}$.

The events may introduce changes in the behaviour of the original EAP. This opens a new problem. In classical action languages we do not care about the previous *history* of the world: If the current state of the world is s, the computation of the resulting states is not affected by the states before s. In the case of EAPs the situation is different, since external updates can change the behaviour of the considered EAP. Fortunately, we do not have to care about the *whole* history of the world, but just about those events containing new initializations, inertial declarations, effects inhibitions, and static and dynamic rules.

It is possible to have a compact description of an EAP that is updated several times via external events. For that we need to further extend the original definition of EAPs.

Definition 10. *An* updated *Evolp action program over* $(\mathcal{F}, \mathcal{A})$ *is any sequence* $I \oplus D_1 \oplus \ldots \oplus D_n$ *where* I *is* ***initialize***$(\mathcal{F})$*, and the various* D_k *are sets consisting of static rules, dynamic rules, inertial declarations and effects inhibitions such that any fluent appearing in* D_k *belongs to* $\mathcal{F}$*.*

Definition 11. *Let* $I \oplus D_1 \oplus \ldots \oplus D_n$ *be any* updated *EAP and* s *a state of the world. Then* s' *is a resulting state from* s *given* $I \oplus D_1 \oplus \ldots \oplus D_n$ *and the sequence of sets of actions* $K_1 \ldots,\ K_n$ *iff there exists an evolving stable model* $M_1, \ldots, M_n$ *of* $I \oplus D_1 \oplus \ldots \oplus D_n$ *given the external events* $(s \cup K_1), \ldots,\ K_n, \emptyset$ *such that* $s' \equiv_{\mathcal{F}} M_n$*.*

In general, if we updated an Evolp action program $I \oplus D$ with the subsequent events $assert(I_1)$, $assert(D_1)$, where $I_1 \oplus D_1$ is another EAP, we obtain the equivalent updated Evolp action program $(I \cup I_1) \oplus D \oplus D_1$ Formally:

Theorem 4. *Let* $I_0 \cup I \oplus D_0 \oplus D_1 \oplus \ldots \oplus D_k$ *be any update EAP over* $(\mathcal{F}, \mathcal{A})$*. Let* $\bigoplus E_i^n$ *be a sequence of events such that:* $E_1 = K_1 \ \cup \ s$*, where* s *is any state of the world and* K_1 *is any set of actions; and the others* E_i*s are any set of actions* K_α*, or any set* $assert($***initialize***$(\mathcal{F}_\beta))$ *where* $\bigcup \mathcal{F}_\beta \equiv I$*, or any* $assert(D_i)$ *with* $1 \leq i \leq k$*. Let* $s_1, \ldots, s_n$ *be a sequence of possible resulting states from* s *given the EAP* $I_0 \oplus D_0$ *and the sequence of events* $\bigoplus E_i^n$ *and* K_{n+1} *a set of actions. Then* $s_1, \ldots, s_n, s'$ *is a resulting state from* s *given* $I_0 \oplus D_0$ *and the sequence of events* $\bigoplus E_i^n \oplus K_{n+1}$ *iff* s' *is a resulting state from* s_n *given* $I_0 \cup I \oplus D_0 \oplus D_1 \oplus \ldots \oplus D_k$ *and the set of actions* K_{n+1}*.*

By applying this theorem we can, for instance, simplify the updates to the original EAP of the example in this section into the updated EAP $I_{sum} \oplus D \oplus D_1 \oplus D_2$, where $I_{sum} \equiv I \ \cup \ I_1 \ \cup \ I_2$, I and D are as in Example 1, and the I_is and D_is are as described above.

Yet one more possibility opened by updated Evolp action programs is to cater for successive elaborations of a program. Consider an initial problem described by an EAP $I \oplus D$. If we want to describe an elaboration of the program, instead of *rewriting* $I \oplus D$ we can simply *update* it with new rules. This gives a new answer to the problem of elaboration tolerance [25] and also open the new possibility of *automatically update* action programs by other action programs.

The possibility to elaborate on an action program is also discussed in [15] in the context of the $\mathcal{C}$ language. The solution proposed there, is to consider $\mathcal{C}$ programs whose rules have one extra fluent atom in their bodies, all these extra fluents being false by default. The elaboration of an action program P is the program $P \cup U$ where U is a new action program. The rules in U can defeat the rules in P by changing the truth value of the extra fluents. An advantage of EAP over that approach is that in EAPs the possibility of updating rules is a built-in feature rather then a programming technique involving manipulation of rules and introduction of new fluents. Moreover, in EAPs we can simply encode the new behaviours of the domain by new rules and then let these new rules update the previous ones.

6 Conclusions and future work

In this paper we have explored the possibility of using logic programs updates languages as action description languages. In particular, we have focused our attention on the Evolp language [2]. As a first point, we have defined a new action language paradigm, christened Evolp action programs, defined as a macro language over Evolp. We have provided an example of usage of this language, and compared Evolp action programs with action languages $\mathcal{A}$, $\mathcal{B}$ and the definite fragment of $\mathcal{C}$, by defining simple translations into Evolp of programs in these languages. Finally, we have also shown and argued about the capability of EAPs to handle changes in the domain during the execution of actions.

Though all the results in this paper refer to the update language Evolp, it is not our stance that these could not be obtained if other LP update languages were used instead. For recasting (some) of the results in other LP update languages, one would have to resort to established relationships between the various LP update languages, such as the ones found in [2, 19]. Also, the possibility of handling changes in the domain shown by EAPs, could in principle be obtained if, instead of Evolp, another update language with the capability of updating update rules were used instead. Another LP update language with this capability is the KABUL language defined in [19]. However, the study of which of the existing LP update languages could be used as action description languages, in a way similar to what is described here for Evolp, is outside the scope of this paper, and would, in our opinion, fit better in a paper with a focus on relationship among various LP update languages. Our goal in this paper was to show that (at least) one LP update language can be used for describing effects of actions, and can be formally compared with existing action description languages. This goal was achieved by showing exactly that for the language Evolp.

Several important topics are not touched here, and will be subject of future work. Important fields of research are how to deal, in the Evolp context, with the problem of planning prediction and postdiction [24], when dealing with incomplete knowledge of the state of the world. Yet another topic involves the possibility of concurrent execution of actions. Nevertheless, we have not fully explored this topic, and confronted the results with extant works [6, 18].

The development of implementations for Evolp and EAPs is another necessary step. Finally EAPs have to be tested in real and complex contexts.

References

1. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. Semantics for dynamic logic programming: a principled based approach. In *7th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, volume 1730 of *LNAI*. Springer, 2004.
2. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *8th European Conf. on Logics in AI (JELIA'02)*, volume 2424 of *LNAI*, pages 50–61. Springer, 2002.
3. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, September/October 2000.
4. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002.
5. J. J. Alferes, L. M. Pereira, T. Przymusinski, H. Przymusinska, and P. Quaresma. Preliminary exploration on actions as updates. In M. C. Meo and M. V. Ferro, editors, *Joint Conference on Declarative Programming (AGP-99)*, 1999.
6. C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31:85–118, 1997.
7. C. Baral, M. Gelfond, and Alessandro Provetti. Representing actions: Laws, observations and hypotheses. *Journal of Logic Programming*, 31, April–June 1997.
8. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming (ICLP-99)*, Cambridge, November 1999. MIT Press.
9. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. In Bernhard Nebel, editor, *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 649–654, San Francisco, CA, 2001. Morgan Kaufmann Publishers, Inc.
10. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of semantics based on causal rejection. *Theory and Practice of Logic Programming*, 2:711–767, November 2002.
11. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
12. M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.
13. M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 16, 1998.
14. E. Giunchiglia, J. Lee, V. Lifschitz, N. Mc Cain, and H. Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.

15. E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153:49–104, 2004.
16. E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI'98*, pages 623–630, 1998.
17. M. Homola. Dynamic logic programming: Various semantics are equal on acyclic programs. In J. Leite and P. Torroni, editors, *5th Int. Ws. On Comptational Logic In Multi-Agent Systems (CLIMA V)*, pages 227–242. Pre-Proceedings, 2004. ISBN: 972-9119-37-6.
18. J. Lee and V. Lifschitz. Describing additive fluents in action language C+. In William Nebel, Bernhard; Rich, Charles; Swartout, editor, *Proc. IJCAI-03*, pages 1079–1084, Cambridge, MA, 2003.
19. J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
20. J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In *LPKR'97: workshop on Logic Programming and Knowledge Representation*, 1997.
21. V. Lifschitz. *The Logic Programming Paradigm: a 25-Year Perspective*, chapter Action languages, answer sets and planning, pages 357–373. Springer Verlag, 1999.
22. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the 3th International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*. Morgan-Kaufmann, 1992.
23. John Wylie Lloyd. *Foundations of Logic Programming*. Springer,, Berlin, Heidelberg, New York,, 1987.
24. J. McCarthy. Programs with commons sense. In *Proceedings of Teddington Conference on The Mechanization of Thought Process*, pages 75–91, 1959.
25. J. McCarthy. *Mathematical logic in artificial intelligence*, pages 297–311. Daedalus, 1988.
26. S. Russel and P. Norvig. *Artificial Intelligence A Modern Approach*. Artificial Intelligence. Prentice Hall, 1995.
27. D. McDermott S. Hanks. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412, (1987).

A Proofs

Before presenting the proofs of the results in this paper, we present an alternative definition of the transition function of EAPs, and prove its equivalence to the original definition (Definition 6). We do so because in some proofs it is more convenient to use this alternative definition.

In this alternative definition, and in its prove, we will use the notation $S|_{\mathcal{I}}$ to denote the restriction of the set S to the literals in the set $\mathcal{I}$ i.e., to denote $S \cap \mathcal{I}$.

Theorem 5. *Let $I \oplus D$ be any EAP, s a state of the world and K a set of actions. Let $\mathcal{R}$ be the set of static rules in D, $\mathcal{I}$ the following set of fluent literals*

$$\mathcal{I} = \{Q \in \mathcal{F} : \ \textbf{\textit{inertial(Q)}} \in D\} \cup \{not\ Q : \ Q \in \mathcal{F} : \ \textbf{\textit{inertial(Q)}} \in D\}$$

and $D(s, K)$ be the following set of rules:

$$D(s, K) = \{\tau : \textbf{\textit{effect}}(\tau) \leftarrow Cond \ \in D \ \wedge K \cup s \models Cond\}$$

Then s' is a resulting state from s given $I \oplus D$ and the set of actions K iff

$$s' = least\left((s \cap s' \cap \mathcal{I}) \ \cup Default(s', \mathcal{R} \cup D(s, K))|_{(\mathcal{F}_L \setminus \mathcal{I})} \ \cup D(s, K) \ \cup \mathcal{R}\right) \tag{6}$$

Proof. By Definition 6, s' is a resulting state from s given $I \oplus D$ and the set of actions K iff there exists an evolving stable model M_1, s^* of $I \oplus D$ given the external events $s \cup K, \emptyset$ such that $s' \equiv_{\mathcal{F}} s^*$. An interpretation M_1 is an evolving stable model of $I \oplus D$ given the external events $s \cup K$ iff M_1 is a refined stable models of $I \oplus D \cup s \cup K$ i.e.,

$$M_1 = least\left((I \cup D \cup s \cup K) \setminus Rej^s(M_1, I \oplus D \cup K \cup s) \cup Default(M_1)\right)$$

All the atoms of the form $event(\tau)$ where τ is the effect of a dynamic rule are false by default in $I \oplus D \cup K \cup s$. Hence the rules of the form (1) and (3), which have those atoms in their bodies, play no role when calculating the least model. Also all the literals of the form $prev(Q)$, where Q is a fluent literal, are false by default, and so the rules of the form $Q \leftarrow prev(Q)$ play no role either. Since the initial (starting) state s is always assumed consistent wrt. the static rules, there is no conflict between the static rules in D. Thus, static rules do not reject any literal in s nor do they infer any fluent literal that does not belong to s. So, we can simplify the expression above in the following way:

$$M_1 = least\left((D^* \cup s \cup K) \cup Default(M_1)\right)$$

where D^* is the set of all rules the form

$$assert(event(\tau)) \leftarrow Cond.$$

for which there is a dynamic rule $\mathbf{effect}(\tau) \leftarrow Cond$ in D, union with the set of all rules of the form

$$assert(prev(Q)) \leftarrow Q. \qquad assert(not\ prev(Q)) \leftarrow not\ Q.$$

for every Q such that $\mathbf{inertial}(Q)$ belongs to D.

Hereafter, for sake of simplicity, in interpretations we omit the negative literals of the form $not\ A$ whenever A is an auxiliary atom or an action literal. In other words, we omit $not\ A$ whenever $A \notin \mathcal{F}$. Moreover, by $Prev(s)$ we denote the set of literals which are either of the form $prev(F)$ where F is a fluent literal that is declared as inertial in D and is true in s, or of the form $not\ prev(F)$ where F is a fluent literal that is declared as inertial in D and is false in s. Finally, by $ED(s, K)$ we mean the set of literals $event(\tau)$ such that

$$assert(event(\tau)) \leftarrow Cond.$$

belongs to D and $s \cup K \models Cond$.

Given this, it is easy to see that the trace associated with any evolving interpretation M_1, s^* is the sequence $\mathcal{J} :\ I \oplus D \oplus Prev(s) \cup ED(s, K)$. So, M_1, s^* is an evolving stable model of $I \oplus D$ given the sequence of events $K, \emptyset$ iff s^* is a refined stable model of $\mathcal{J}$.

Let s^* be any interpretation over the language of $I \oplus D$, and $s' = s^*|_{\mathcal{F}}$. To prove the theorem, we simply have to prove that s^* is a refined stable model of $\mathcal{J}$ iff s' satisfies the equivalence (6). By definition of refined stable model, s^* is a refined stable model of $\mathcal{J}$ iff

$$s^* = least\left((I \cup D \cup Prev(s) \cup D(s, K)) \setminus Rej^S(s^*) \cup Default(s^*)\right)$$

$\Rightarrow$ Assume that s^* is a refined stable model of $\mathcal{J}$. To prove that s' satisfies the equivalence, we start by simplifying the expression above defining s^*.
Let $s' = s^*_{\mathcal{F}}$. Since s' only has fluent literals, the dynamic rules and the inertial declarations in D play no role in verifying the equivalence. Hence, the only rules we are interested in are the static rules in $\mathcal{R}$. Moreover, since s^* is two valued, there is no mutual rejection between the rules in $\mathcal{R}$: otherwise there would be a fluent literal Q such that all the rules with head Q or $not\ Q$ would be rejected, and such that $not\ Q$ would not be in the set $Default(s^*)$ as well. In such a case, neither Q nor $not\ Q$ would be in s^* which would contradict the two valuedness of s^*. Finally, by partially evaluating the facts in $ED(s, K)$, in the rules of the form

$$F \leftarrow Body,\ event(F \leftarrow Body).$$

we can delete the atoms $event(\tau)$ from the body of those rules whenever $event(\tau) \in ED(s, K)$, and delete one such rule when $event(\tau) \notin ED(s, K)$. With this, we can simplify the equivalence for s' into:

$$s' = least\left(I \setminus Rej^S(s^*) \cup Prev(s) \cup \mathcal{R} \cup D(s, K) \cup Default(s^*)\right)$$

We can split the set of default assumptions into two subsets: the one concerning the inertial fluent literals; and the one concerning the fluent literals that are not inertial. Taking this splitting in consideration, the equivalence for s' becomes:

$$s' = least \begin{pmatrix} I \setminus Rej^S(s^*) \cup Prev(s) \cup Default(s^*)|_{\mathcal{I}} \cup \\ \mathcal{R} \cup D(s,K) \cup Default(s^*)|_{(\mathcal{F}_L \setminus \mathcal{I})} \end{pmatrix}$$

where $Default(s^*)$ stands for $Default(s^*, I \oplus R \cup D(s,K))|_{(\mathcal{F}_L \setminus \mathcal{I})}$. Notice that the expression $Default(s^*, I \oplus R \cup D(s,K))|_{(\mathcal{F}_L \setminus \mathcal{I})}$ is equivalent to $Default(s', \mathcal{R} \cup D(s,K))|_{(\mathcal{F}_L \setminus \mathcal{I})}$. Moreover, the expression $Default(s^*)|_{\mathcal{I}}$ is equivalent to $Default(s', s \cup \mathcal{R} \cup D(s,K))|_{(\mathcal{I})}$. Let $Inherit(s)$ be the set of rules:

$$Inherit(s^*) = \{Q \in \mathcal{F} : Q \leftarrow prev(Q) \in I \setminus Rej^S(s^*) \wedge prev(Q) \in Prev(s)\}$$

What remains to show in order to prove that s' satisfies the equivalence (6) is that

$$Inherit(s^*) \cup Default(s^*)|_{\mathcal{I}} \equiv (s \cap s' \cap \mathcal{I})$$

For showing this, we consider separately the negative and the positive fluent literals. Let Q be a fluent literal that belongs to $(s \cap s' \cap \mathcal{I})$. We want to prove this is equivalent to say that $Q \leftarrow prev(Q)$ belongs to $I \setminus Rej^S(s^*)$ and that $Prev(Q) \in Prev(s)$ i.e., we want to prove that $Q \in Inherit(s^*)$. The literal Q belongs to $(s \cap s' \cap \mathcal{I})$ iff $Q \in \mathcal{I}$, *not* $Q \notin s$ and *not* $Q \notin s'$. This implies that there exists no rule in $\mathcal{R} \cup D(s,K)$ whose head is *not* Q and whose body is true. So, the rule $Q \leftarrow prev(Q)$ belongs to $I \setminus Rej^S(s^*)$ and, by $Q \in s$ and by definition of $Prev(s)$, we conclude that $Prev(Q) \in Prev(s)$. Let assume now $Q \leftarrow prev(Q)$ belongs to $I \setminus Rej^S(s^*)$, then there exists no rule in $\mathcal{R} \cup D(s,K)$ whose head is *not* Q and whose body is true. If, furthermore, $Prev(Q) \in Prev(s)$, then *not* $Q \notin Default(s^*)$ and so *not* Q is not derived by any rule nor by default assumption. Thus, *not* $Q \notin s'$ and so $Q \in s$. Moreover, by definition if $prev(Q) \in Prev(s)$ then $Q \in s$ and $Q \in \mathcal{I}$. So, we have proved that

$$Q \in (s \cap s' \cap \mathcal{I}) \Leftrightarrow Q \leftarrow prev(Q) \in I \setminus Rej^S(s^*) \ \wedge \ prev(Q) \in Prev(s)$$

Let us now consider the negative fluent literals. In this case we want to prove that, for any inertial fluent, the following equivalence holds.

$$not\ Q \in (s \cap s') \Leftrightarrow not\ Q \in Default(s', s \cup \mathcal{R} \cup D(s,K))|\mathcal{F}$$

We know *not* $Q \in s'$ iff $Q \notin s'$, which, since s' is a model of $\mathcal{R} \cup D(s,K)$, implies that there exists no rule in $\mathcal{R} \cup D(s,K)$ whose head is Q and whose body is satisfied by s'. This, together with the fact that $Q \notin s$, by definition of $Default$ implies that *not* $Q \in Default(s', s \cup \mathcal{R} \cup D(s,K))$, as desired.

$\Leftarrow$ Let us now suppose that s' satisfies the equivalence (6). i.e.

$$s' = least\left((s \cap s' \cap \mathcal{I}) \ \cup Default(s', \mathcal{R} \cup D(s,K))|_{(\mathcal{F}_L \setminus \mathcal{I})} \ \cup D(s,k) \ \cup \mathcal{R}\right)$$

Let NED be the set of literals of the form $\neg event(\tau)$ such that $event(\tau) \in ED(s,K)$ and there is no dynamic rule of the form $\mathbf{effect(\tau)} \leftarrow Cond$ such that s' satisfies $Cond$. Let s' be the following evolving interpretation (again we omit in the interpretation, the negative literals which are not fluent literals).

$$s^* = s' \ \cup \ Prev(s) \ \cup \ ED(s,K) \ \cup NED \cup assert(ED(s',K)) \cup \\ \cup \ assert(Prev(s))'$$

We have to prove that s^* is a refined stable model of $\mathcal{J}$. We start this proof by showing that

$$Inherit(s^*) \cup Default(s^*)|_{\mathcal{I}} \equiv (s \cap s' \cap \mathcal{I})$$

We start by assuming that Q is a fluent literal in $(s \cap s' \cap \mathcal{I})$. Q is such a fluent iff $Prev(Q) \in Prev(s)$, and $not\ Q \notin s'$. Since s' is a model of $\mathcal{R} \cup D(s,K)$, we conclude that there exists no rule in $\mathcal{R} \cup D(s,K)$ with head $not\ Q$ and true body in s'. Thus, the rule $Q \leftarrow prev(Q) \in I \setminus Rej^S(s^*)$, and hence $Q \in Inherit(s^*)$.
Let assume now $Q \in Inherit(s^*)$ (i.e. $Q \leftarrow prev(Q) \in I \setminus Rej^S(s^*)$ and $prev(Q) \in Prev(s)$) then $Q \in s$. This implies that $not\ Q \notin s$, $Q \in \mathcal{I}$, and there exists no rule in $\mathcal{R} \cup D(s,K)$ with head Q whose body is true in s'. Consequently, $not\ Q \notin s'$ (i.e. $Q \in s'$), and finally $Q \in (s \cap s' \cap \mathcal{I})$.
Let us now consider the negative fluent literals. We want to prove that, for any inertial fluent, the following equivalence holds.

$$not\ Q \in (s \cap s') \Leftrightarrow not\ Q \in Default(s', s \cup \mathcal{R} \cup D(s,K))|\mathcal{F}$$

The proof proceeds in the same way as above, in order to conclude that

$$Inherit(s^*) \cup Default(s^*)|_{\mathcal{I}} \equiv (s \cap s' \cap \mathcal{I})$$

We obtain then the following equivalence

$$s' = least \begin{pmatrix} Inherit(s^*) \cup Default(s^*)|_{\mathcal{I}} \cup \\ Default(s', \mathcal{R} \cup D(s,K))|_{(\mathcal{F}_L \setminus \mathcal{I})} \ \cup D(s,k) \ \cup \mathcal{R} \end{pmatrix}$$

which is equivalent to

$$s' = least\,(\ Inherit(s^*) \cup Default(s^*) \ \cup D(s,k) \ \cup \mathcal{R})\,|_{\mathcal{F}_L}$$

Since s' is consistent wrt. $D(s,K)$ and $\mathcal{R}$, these sets of rules do not contain any pair of rules with conflicting heads and whose bodies are both true in s'. So, by replacing $Inherit(s^*)$ with $Prev(s) \cup I \setminus Rej^S(s^*)$ we obtain

$$s' = least\,\big(\ (I \cup D(s,K) \cup mR) \setminus Rej^S(s^*) \cup Default(s^*)\big)\,|_{\mathcal{F}_L}$$

and from this, and by considering the definition of s^*

$$s^* = least\,\big((I \cup D \cup Prev(s) \cup D(s,K)) \setminus Rej^S(s^*) \cup Default(s^*)\big)$$

This equation is, by definition, equivalent to say that M_1, s^* is an evolving stable model of $I \oplus D$ given the sequence of events $K, \emptyset$. In other words, s' is a resulting state from s given $I \oplus D$ and the set of actions K.

In the extreme cases where the set of inertial fluents coincides with the whole set of fluents and, when the set if inertial fluents is empty, we obtain two simplifications of the equivalence (6).

Corollary 1. *Let $I \oplus D$ be any EAP, s a state of the world and K a set of actions. Let $\mathcal{R}$, $D(s,K)$ be as in theorem 5. Moreover let every fluent be an inertial fluent. Then s' is a resulting state from s given $I \oplus D$ and the set of actions K iff*

$$s' = least\,(s \cap s') \ \cup D(s,k) \ \cup \mathcal{R})$$

Proof. Follows trivially as a special case of theorem 5.

Corollary 2. *Let $I \oplus D$ be any EAP, s a state of the world and K a set of actions. Let $\mathcal{R}$, $D(s,K)$ be as in theorem 5. Moreover let the set of inertial fluents be the empty set. Then s' is a resulting state from s given $I \oplus D$ and the set of actions K iff s' is a stable model of the logic program $D(s,k) \ \cup \mathcal{R}$*

Proof. It follows trivially as a special case of theorem 5 that

$$s' = least\left(Default(s', \mathcal{R} \cup D(s,K))|_{(\mathcal{F}_L \setminus \mathcal{I})} \ \cup D(s,k) \ \cup \mathcal{R}\right)$$

As proved in [19] this amount to say s' is a stable model of $D(s,k) \ \cup \mathcal{R}$.

Having shown this alternative to the definition of the transition function of EAPs, and proven its equivalence to the original Definition 6, we are now ready to prove all of the theorems (that we recall here, for the sake of readability) in this paper.

Theorem 1 (Complexity of EAPs). *Let $I \oplus D$ be any EAP over $(\mathcal{F}, \mathcal{A})$, s a state of the world and $K \subseteq \mathcal{A}$. To find a resulting state s' from s given $I \oplus D$ and the set of actions K is an NP-complete problem.*

Proof. By corollary 2, and given that the problem of finding a stable model of a program is NP-hard, we conclude that finding a resulting state s' from s given $I \oplus D$ and the set of actions K is an NP-hard problem.

As for membership, from theorem 5 and from the observation that the computation of $least(P)$, where P is a logic program, is polynomial wrt. the number of rules in P (since $least(P)$ is the least Herbrand model of P considering the negative literals in P as new atoms), it follows that checking whether a given state s' is resulting state is a polynomial problem wrt. the number of rules in $I \oplus D$ plus the number of elements in $\mathcal{F} \cup \mathcal{A}$. Hence, the problem of finding a resulting state s' from s given $I \oplus D$ and the set of actions K is NP.

Theorem 2 (Relation to $\mathcal{B}$). *Let P be any $\mathcal{B}$ program with set of fluents $\mathcal{F}$, $(I^B \oplus D^{BP}, \mathcal{F})$ its translation, s a state and K any set of actions. Then s' is a resulting state from s given P and the set of actions K iff it is a resulting state from s given $I^B \oplus D^{BP}$ and the set of actions K.*

Proof. It trivially follows from corollary 1.

Theorem 3 (Relation to $\mathcal{C}$). *Let P be any action program in the definite fragment of $\mathcal{C}$ with set of fluents $\mathcal{F}$, $(I^C \oplus D^{CP}, \mathcal{F}^C)$ its translation, s a state, s^C the interpretation over $\mathcal{F}^C$ defined as follows: $s^C = s \cup \{Q_N \mid Q \in s\} \cup \{not\ Q_N \mid not\ Q \in s\}$ and K any set of actions. Then s^* is a resulting state from s^C given $I^C \oplus D^{CP}$ and the set of actions K iff there exists s' such that s' is a resulting state from s, given P and the set K and $s^* \equiv_{\mathcal{F}_L} s'$.*

Proof. By corollary 2, s^* is a resulting state from s^C given $I^C \oplus D^{CP}$ and the set of actions K iff s' is a stable model of the program $\mathcal{R} \cup D(s, K)$ where $\mathcal{R}$ and $D(s^C, K)$ are defined as in theorem 5. From the translation of definite causal theories into logic programs presented in [15], it follows that this is equivalent to say that s' is a model of the causal theory obtained by all the static rules of P plus the rules of the form **caused** J **if** H for which a dynamic rule

$$\textbf{caused } J \textbf{ if } H \textbf{ after } O$$

belongs to P and Q is true in $s \cup K$. This, in turn, is equivalent to saying that s' is a resulting state from s given P and the set of actions K, as desired.

Theorem 4 (Simplification of updated EAPs). *Let $I_0 \cup I \oplus D_0 \oplus D_1 \oplus \ldots \oplus D_k$ be any update EAP over $(\mathcal{F}, \mathcal{A})$. Let $\bigoplus E_i^n$ be a sequence of events such that: $E_1 = K_1 \cup s$, where s is any state of the world and K_1 is any set of actions; and the others E_is are any set of actions K_α, or any set $assert(\textbf{\textit{initialize}}(\mathcal{F}_\beta))$ where $\bigcup \mathcal{F}_\beta \equiv I$, or any $assert(D_i)$ with $1 \leq i \leq k$. Let $s_1, \ldots, s_n$ be a sequence of possible resulting states from s given the EAP $I_0 \oplus D_0$ and the sequence of events $\bigoplus E_i^n$ and K_{n+1} a set of actions. Then $s_1, \ldots, s_n, s'$ is a resulting state from s given $I_0 \oplus D_0$ and the sequence of events $\bigoplus E_i^n \oplus K_{n+1}$ iff s' is a resulting state from s_n given $I_0 \cup I \oplus D_0 \oplus D_1 \oplus \ldots \oplus D_k$ and the set of actions K_{n+1}.*

Proof. The sequence $s_1, \ldots, s_n, s'$ is a sequence of possible resulting states iff there exists a sequence of evolving interpretations $M_0, M_1, \ldots M_n, s^*$ such that $M_0|_{\mathcal{F}} \equiv s$, $M_i|_{\mathcal{F}} \equiv s_i$ and $s^*|_{\mathcal{F}} \equiv s'$. The trace of $M_0, M_1, \ldots M_n, s^*$ is the DLP $I_0 \oplus D_0 \oplus T_1 \ldots \oplus T_n$ where each T_is is a set of literal of one of the following forms:

$T_i = Aux_i$
$T_i = Aux_i \cup \textbf{initialize}(\mathcal{F}_\beta)$
$T_i = Aux_i \cup D_j$ for some $0 \leq j \leq k$

and Aux_i is a set of auxiliary literals of the form $Prev(Q)$ or $not\ Prev(Q)$, where Q is an inertial literal or $event(\tau)$ or $not\ event(\tau)$, τ being the effect of some dynamic rule.

To compute s^*, the only relevant part of the trace is formed by the various **initialize**$(\mathcal{F}_\beta s)$, $D_k s$ and the last set of auxiliary literals Aux_n. Moreover, the

semantics does not change if we put the various **initialize($\mathcal{F}_\beta s$)** in the first program of the sequence, since a fluent only appears in a D_j after being initialized. Hence we can simplify the trace of $M_0, M_1, \ldots M_n, s^*$ into:

$$I_0 \cup I \oplus D_0 \oplus D_1 \oplus \ldots \oplus D_k \cup Aux_n$$

The set Aux_n can be split in three separate sets

$$Aux_n = Prev(s_n) \cup ED(s_n, K) \cup Retract(s_n)$$

where $Prev(s_n)$ and $ED(s_n, K)$ are as defined in the proof of theorem 5 and $Retract(s_n)$ is the set of all literals of the form $not\ event(\tau)$ coming from dynamic rules whose preconditions are true in s_{n-1} and false in s_n. The negative literals in $Retract(s_n)$ simply rejects facts of the form $event(\tau)$ from Aux_{n-1}. Since we have already simplified the trace by erasing all the $Aux_i s$ with $i < n$, we can ignore the set $Retract(s_n)$. Thus, we obtain that $s_1, \ldots s'$ is a sequence of possible resulting states iff an interpretation s^*, with $s^*|_{\mathcal{F}_L} \equiv s'$, is a refined stable model of $I_0 \cup I \oplus D_0 \oplus D_1 \oplus \ldots \oplus D_k \oplus ED(s_n, K) \cup Prev(s_n)$. This is equivalent to saying that s' is a resulting state from s given $I_0 \cup I \oplus D_0 \oplus D_1 \oplus \ldots \oplus D_k$ and the set of actions K_{n+1}, as desired.